Applications of Linear Types

Dhruv C. Makwana
Trinity College College

A dissertation submitted to the University of Cambridge
in partial fulfilment of the requirements for the degree of
Master of Engineering in Part III of the Computer Science Tripos

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: dcm41@cam.ac.uk

April 18, 2018

Declaration

I Dhruv C. Makwana of Trinity College College, being a candidate for Computer Science Tripos, Part III, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 0

Signed:

Date:

Abstract

This is the abstract. Write a summary of the whole thing. Make sure it fits in one page.

Contents

List of Figures

List of Tables

1 Introduction

This is the introduction where you should introduce your work. In general the thing to aim for here is to describe a little bit of the context for your work — why did you do it (motivation), what was the hoped-for outcome (aims) — as well as trying to give a brief overview of what you actually did.

It's often useful to bring forward some "highlights" into this chapter (e.g. some particularly compelling results, or a particularly interesting finding).

It's also traditional to give an outline of the rest of the document, although without care this can appear formulaic and tedious. Your call.

Leaving this chapter until last.

1.1 Overview of Problem

1.2 Contributions

2 | Background

In this chapter, I will outline the concept of linear types and show how they can be used to solve the problems faced by programmers writing code using linear-algebra libraries. I will be emphasising the *practical* and intuitive explanations of linear types to keep this thesis accessible to working programmers as well as academics not familiar with type-theory; giving only a terse overview of the history and theory behind linear types for the interested reader to pursue further.

2.1 Tracking Resources with Linearity

Familiar examples of using a type-system to express program-invariants are existential-types for abstraction and encapsulation, polymorphic types for parametricity and composition (a.k.a generics). Less-known examples include dependent-types (contracts or pre- and post-conditions). The advantages of using a type-system to express program invariants are summarised by saying the stronger the rules you follow, the better the guarantees you can get about your program, *before* you run it. At first, the rules seems restrictive, but similar to how the rules of grammar, spelling and more generally writing help a writer make it easier and clearer to communicate the ideas they wish to express, so to do rules about typing make it easier to communicate the intent and assumptions under which a program is written. An added, but often overlooked benefit is automated-checking: a programmer can boldly refactor in certain ways and the compiler will *assist* in ensuring the relevant invariants the type-system enforces are updated and kept consistent by pointing-out where they are violated.

Linear types are a way to help a programmer track and manage resources. In practical programming terms, they enforce the restriction that a value may be used exactly once.[1] While this restriction may seem limiting at first, precisely these constraints can be used to express common invariants of the programs written by working programmers every day. For example: a file or a socket, once opened *must* closed; all memory that is manually allocated must be freed. C++'s destructors and Rust's Drop-trait (and more generally, its borrow-checker) attempt to enforce these constraints by basically doing the same thing: any resource that has not been moved is deallocated at the end of the current lexical scope. Notably, these languages also permit aliasing, alongside rules enforcing when it is acceptable to do so. On face value, the above one-line description of linear types prevents aliasing or functions such as $\lambda x.\ x \times x$, such features are still allowed (albeit in a more restricted fashion) in a *usable* linear type system designed for working with linear-algebra libraries.

[1] This definition may differ from more colloquial uses in discussions surrounding *substructural* type systems and/or Rust. A brief explanation can be found in Section 2.4.

2.2 Problem in Detail

Given this background, the most pertinent question at hand is: what problems do linear-algebra library users (and writers) typically face? The answer to this question depends on which of two buckets a programmer falls (or is forced by domain) into. On one side, we have users of high-level linear-algebra libraries such as Owl (for OCaml), Julia and Numpy (for Python); other the other, we have users of more manual, lower-level libraries such as BLAS (Basic Linear Algebra Subroutines) and Eigen for languages like C++ and FORTRAN.[2]

2.2.1 One Too Many Copies and a Thousand Bytes Behind

2.2.2 IHNIWTNM

The title of this subsection[3] illustrates the problem with the C++/FORTRAN side: legibility (and ease of reasoning) is sacrified at the alter of performance and efficiency.

2.3 Proposed Solution

A (E)DSL!

2.4 Further Reading and Theory

- History - Girard and so on
- Congratulations, it's a co-monad
- Co-effects
- General direction of field - Haskell, Rust, RAML, Atkey/McBride & Idris, Krishnaswami

[2]I am not including Rust in this comparison because its linear-algebra libraries are under active development and not as well-known/used. Later on, given that it is a language with in-built support of substructural features to track resources, Rust will be compared and constrasted with this project to evaluate the classic (E)DSL-versus-language-feature debate as it applies to the domain of linear-algebra libraries.

[3]I Have No Idea What Those Names Mean.

3 | Implementation

This chapter may be called something else... but in general the idea is that you have one (or a few) "meat" chapters which describe the work you did in technical detail.

I implemented this project in OCaml, however I strongly believe the ideas described in this chapter can be applied easily to other languages and also are modular enough to extend the OCaml implementation to output to different back-end languages. I will show how a small core language with a few features can be the target of heavy-desugaring of typical linear-algebra programs. This core language can then be elaborated and checked for linearity before performing some simple and predictable optimisations and emitting (in this particular implementation) OCaml code that is not obviously safe, correct and performant.

3.1 Core Language

3.2 Elaboration and Inference

3.3 Compiling and Metaprogramming

4 Evaluation

For any practical projects, you should almost certainly have some kind of evaluation, and it's often useful to separate this out into its own chapter.

In this chapter, I will argue the central premise of this thesis: linear types are a practical and usable tool to help working programmers write code that is more legible and less resource-hungry than with existing linear-algebra frameworks. My project illustrates how to do so in a way that can be implemented as a usable *library* for existing languages and frameworks that leverages the already impressive amount of work gone into optimising them so far, whilst overcoming some of their shortcomings.

4.1 Against Overly-Safe Copying

4.2 Against Poor Legibility

5 | Related Work

This chapter covers relevant (and typically, recent) research which you build upon (or improve upon). There are two complementary goals for this chapter:

1. to show that you know and understand the state of the art; and
2. to put your work in context

Ideally you can tackle both together by providing a critique of related work, and describing what is insufficient (and how you do better!)

The related work chapter should usually come either near the front or near the back of the dissertation. The advantage of the former is that you get to build the argument for why your work is important before presenting your solution(s) in later chapters; the advantage of the latter is that don't have to forward reference to your solution too much. The correct choice will depend on what you're writing up, and your own personal preference.

Now that I have described my project and thesis, I will explain how it relates to existing work, with brief discussions on possible extensions in each case.

6 | Conclusion

As you might imagine: summarizes the dissertation, and draws any conclusions. Depending on the length of your work, and how well you write, you may not need a summary here.

You will generally want to draw some conclusions, and point to potential future work.

Bibliography